Levels of representation and abstraction

A digital system can be represented at different levels of abstraction [1]. This keeps the description and design of complex systems manageable. Figure 1 shows different levels of abstraction.

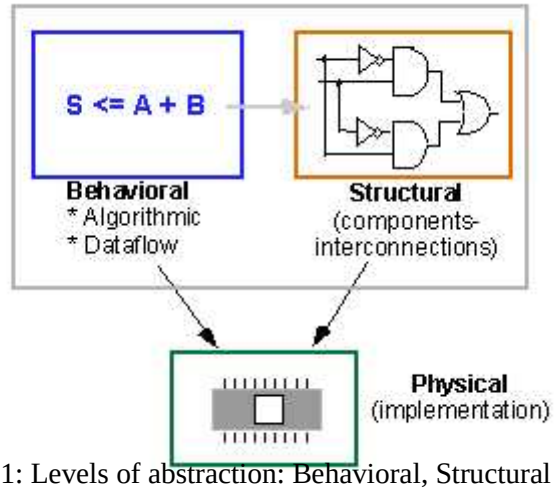

Figure 1: Levels of abstraction: Behavioral, Structural and Physical

The highest level of abstraction is the **behavioral** level that describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them. A behavioral description specifies the relationship between the input and output signals. This could be a Boolean expression or a more abstract description such as the Register Transfer or Algorithmic level. As an example, let us consider a simple circuit that warns car passengers when the door is open or the seatbelt is not used whenever the car key is inserted in the ignition lock At the behavioral level this could be expressed as,

Warning = Ignition_on AND (Door_open OR Seatbelt_off)

The **structural** level, on the other hand, describes a system as a collection of gates and components that are interconnected to perform a desired function. A structural description could be compared to a schematic of interconnected logic gates. It is a representation that is usually closer to the physical realization of a system. For the example above, the structural representation is shown in Figure 2 below.

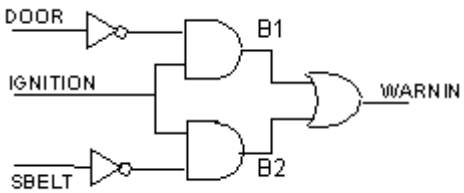

figure 2: Structural representation of a “buzzer” circuit.

VHDL allows one to describe a digital system at the structural or the behavioral level. The behavioral level can be further divided into two kinds of styles: **Data flow** and **Algorithmic**. The dataflow representation describes how data moves through the system. This is typically done in terms of data flow between registers (Register Transfer level). The data flow model makes use of concurrent statements that are executed in parallel as soon as data arrives at the input. On the other hand, sequential statements are

executed in the sequence that they are specified. VHDL allows both concurrent and sequential signal assignments that will determine the manner in which they are executed. Examples of both representations will be given later.

Basic Structure of a VHDL file

A digital system in VHDL consists of a design **entity** that can contain other entities that are then considered components of the top-level entity. Each entity is modeled by an *entity declaration* and an *architecture body*. One can consider the entity declaration as the interface to the outside world that defines the input and output signals, while the architecture body contains the description of the entity and is composed of interconnected entities, processes and components, all operating concurrently, as schematically shown in Figure 3 below. In a typical design there will be many such entities connected together to perform the desired function.

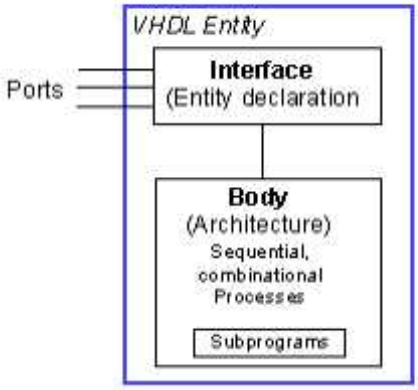

Figure 3: A VHDL entity consisting of an interface (entity declaration) and a body (architectural description).

VHDL uses reserved **keyword**s that cannot be used as signal names or identifiers. Keywords and user-defined identifiers are **case insensitive**. Lines with comments start with two adjacent hyphens (--) and will be ignored by the compiler. VHDL also ignores line breaks and extra spaces. VHDL is **a strongly typed** language which implies that one has always to declare the type of every object that can have a value, such as signals, constants and variables.

a. Entity Declaration

The entity declaration defines the NAME of the entity and lists the input and output ports. The general form is as follows,

entity NAME_OF_ENTITY **is** [**generic** *generic_declarations*);]

port (*signal_names*: **mode** *type*;

signal_names: **mode** *type*;

:

signal_names: **mode** *type*);

end [NAME_OF_ENTITY] ;

An entity always starts with the keyword entity, followed by its name and the keyword is. Next are the port declarations using the keyword **port**. An entity declaration always ends with the keyword **end**, optionally [] followed by the name of the entity.

- The NAME_OF_ENTITY is a user-selected identifier
    - signal_names consists of a comma separated list of one or more user-selected identifiers that specify external interface signals.
    - **mode**: is one of the reserved words to indicate the signal direction:
        - **in** – indicates that the signal is an input
        - **out** – indicates that the signal is an output of the entity whose value can only be read by other entities that use it.
        - **buffer** – indicates that the signal is an output of the entity whose value can be read inside the entity's architecture
        - **inout** – the signal can be an input or an output.
    - *type*: a built-in or user-defined signal type. Examples of types are bit, bit_vector, Boolean, character, std_logic, and std_ulogic.
        - *bit* – can have the value 0 and 1
        - *bit_vector* – is a vector of bit values (e.g. bit_vector (0 to 7)
        - *std_logic, std_ulogic, std_logic_vector, std_ulogic_vector*: can have 9 values to indicate the value and strength of a signal. Std_ulogic and std_logic are preferred over the bit or bit_vector types.
        - *boolean* – can have the value TRUE and FALSE
        - *integer* – can have a range of integer values
        - *real* – can have a range of real values
        - *character* – any printing character
        - *time* – to indicate time

- **generic:** generic declarations are optional and determine the local constants used for timing and sizing (e.g. bus widths) the entity. A generic can have a default value. The syntax for a generic follows,

generic (

constant_name: type [:=value] ;

constant_name: type [:=value] ;

:

constant_name: type [:=value]);

Architecture body

The architecture body specifies how the circuit operates and how it is implemented. As discussed earlier, an entity or circuit can be specified in a variety of ways, such as behavioral, structural (interconnected components), or a combination of the above.

The architecture body looks as follows,

architecture architecture_name **of** NAME_OF_ENTITY **is**

-- Declarations

-- components declarations

-- signal declarations

-- constant declarations

-- function declarations

-- procedure declarations

-- type declarations

begin

-- Statements

:

end architecture_name;

Library and Packages: **library** and **use** keywords

A library can be considered as a place where the compiler stores information about a design project. A VHDL package is a file or module that contains declarations of commonly used objects, data type, component declarations, signal, procedures and functions that can be shared among different VHDL models.

We mentioned earlier that std_logic is defined in the package ieee.std_logic_1164 in the ieee library. In order to use the std_logic one needs to specify the library and package. This is done at the beginning of the VHDL file using the **library** and the **use** keywords as follows:

library ieee;

use ieee.std_logic_1164.**all**;

The .**all** extension indicates to use all of the ieee.std_logic_1164 package.

The Xilinx Foundation Express comes with several packages.

ieee Library:

std_logic_1164 package: defines the standard datatypes

- std_logic_arith package: provides arithmetic, conversion and comparison functions for the signed, unsigned, integer, std_ulogic, std_logic and std_logic_vector types
- std_logic_unsigned
- std_logic_misc package: defines supplemental types, subtypes, constants and functions for the std_logic_1164 package.

To use any of these one must include the library and use clause:

library ieee;

use ieee.std_logic_1164.**all**;

use ieee.std_logic_arith.**all**;

use ieee.std_logic_unsigned.**all**;

In addition, the synopsis library has the attributes package:

library SYNOPSYS;

use SYNOPSYS.attributes.**all**;

One can add other libraries and packages. The syntax to declare a package is as follows:

```
-- Package declaration

package name_of_package is

        package declarations

end package name_of_package;

-- Package body declarations
package body name_of_package is

        package body declarations

end package body name_of_package;
```

For instance, the basic functions of the AND2, OR2, NAND2, NOR2, XOR2, etc. components need to be defined before one can use them. This can be done in a package, e.g. basic_func for each of these components, as follows:

```
-- Package declaration

library ieee;

use ieee.std_logic_1164.all;

package basic_func is

        -- AND2 declaration

        component AND2

                generic (DELAY: time :=5ns);

                port (in1, in2: in std_logic; out1: out std_logic);

        end component;

        -- OR2 declaration

        component OR2

                generic (DELAY: time :=5ns);

                port (in1, in2: in std_logic; out1: out std_logic);

        end component;
```

```
end package basic_func;

-- Package body declarations

library ieee;

use ieee.std_logic_1164.all;

package body basic_func is

	-- 2 input AND gate
	entity AND2 is

		generic (DELAY: time);

		port (in1, in2: in std_logic; out1: out std_logic);

	end AND2;

	architecture model_conc of AND2 is

		begin

			out1 <= in1 and in2 after DELAY;

	end model_conc;

	-- 2 input OR gate
	entity OR2 is

		generic (DELAY: time);

		port (in1, in2: in std_logic; out1: out std_logic);

	end OR2;

	architecture model_conc2 of AND2 is

		begin

			out1 <= in1 or in2 after DELAY;

	end model_conc2;

end package body basic_func;
```